EFFICIENT PROGRAM SYNTHESIS IN COMPUTATIONAL MODELS

MAX I. KANOVICH

▷ Starting from the system PRIZ, the method of automatic program synthesis is used in AI systems. We determine (1) the precise description of a class of computational models; (2) the precise description (acute semantics) of such concepts as "a solution of a computational task" and "solvability of a computational task"; (3) the calculus of dependencies OLD, which is proved to be correct and strongly complete with respect to the proposed semantics. On the basis of OLD, we construct an algorithm which analyzes a computational task and synthesizes a program for its solution and which runs in small (subquadratic) space with quasi polynomial time (the degree of the polynomial does not exceed the minimal number of subtasks that must interact in any solution of the main task). This algorithm can synthesize optimal programs which have minimal execution time. ◁

1. INTRODUCTION

Starting from the system PRIZ, the method of automatic program synthesis is used for AI systems [1–11].

The problem of program synthesis is formulated in the following way. A specification called the computational model M is given. A system of relations KB (the so-called knowledge base) is the core of the model M. Tasks of the form

"on the model M find Z from X"

are proposed.

One would like

(1) to analyze the task: determine whether the *functional dependency* $(X \to Z)$ follows from the relations of KB;

Address correspondence to Max I. Kanovich, Kalinin State University, Kalinin 170000, U.S.S.R.
Received January 1989.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/90/$3.50

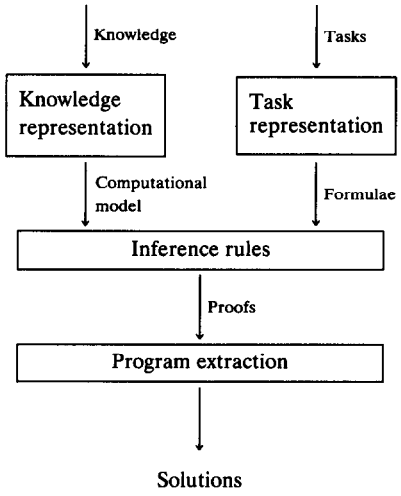

FIGURE 1

(2) to synthesize a solution of the task: using the relations of KB, to compose a program which, from given values of inputs of the list X, computes the values of outputs of the list Z.

The problem is reduced to theorem proving in certain formal theories.

The general scheme is shown in Figure 1.

The system of knowledge representation creates a computational model M and its core—the knowledge base KB. The task representation gives a task as a formula of a special kind. On the basis of the inference rules one is looking for a proof from a given set of *axioms* describing the computational model and the computational task. In the case of success a program is extracted from the proof.

Unfortunately, such a problem of analysis and synthesis in general is unsolvable. Therefore these intellectual systems should be oriented to special classes of computational models and computational tasks.

In theoretical foundations of such AI systems at least five problems arise:

(1) Precise description of a class of computational models considered: in particular, one should determine

(a) the types of entities in the entity system Ent,
(b) the forms of relations in the knowledge base KB.

(2) Precise determination (acute semantics) of the concepts:

"a solution of a computational task",

"solvability of a computational task".

In particular, the question arises how to explain that the computational task is unsolvable in an independent manner without reference to a known algorithm of analysis and synthesis or to a system of inference rules which has been proposed before.

(3) Verification of the set of inference rules with respect to this semantics, i.e., whether a program extracted from a proof is correct.

(4) The problem of the completeness of the system of inference rules with respect to the semantics: whether the given system of inference rules provides sufficient abilities to obtain a solution of every computational task which is solvable with respect to the proposed semantics.

(5) The problem of the constructivity of the system of inference rules: whether one can extract a program from a proof in the system.

In practical realizations of such AI systems at least five additional problems arise:

(1) Making the knowledge representation language clear and simple.

(2) Proposing an efficient strategy for searching for proofs (at least of a certain kind) in the given calculus.

(3) Building the program extraction procedure.

(4) Proposing an efficient (with respect to time and space) algorithm for the analysis of computational tasks and the synthesis of its solutions (at least for "natural" tasks).

(5) Finally, guaranteeing the quality of programs synthesized, and proposing an algorithm (efficient with respect to time and space) for the synthesis of optimal programs.

It should be pointed out that the problem of searching for proofs in formal systems is the main obstacle to the practical realization of such AI systems.

General theorems of complexity theory show that the problem of analyzing a computational task and synthesizing a program for its solution is algorithmically unsolvable. Even when we restrict the class of computational models, all the theoretical troubles concerning NP- and SPACE-completeness arise. On the other hand, there exists wide experience in the practical use of the system PRIZ, which synthesizes programs for practical computational tasks successfully and efficiently. One of the stimuli for writing this paper was the desire to understand this strange phenomenon, that a certain AI system can work successfully and efficiently in practice, when its complexity is proved to have exponential uniform lower estimate.

In this paper, on the basis of a specialized calculus, we construct algorithms which can synthesize programs on a wide class of computational models in quasipolynomial time.

2. COMPUTATIONAL MODELS

We investigate computational models for which

(1) entities in the model can be of both simple and functional types (functionals of higher orders are permitted); the domain of a functional entity is the set of all programs of a particular type;

(2) the knowledge bass KB can contain functional, operator, circular, and variant dependencies.

2.1. *Formal Definition*

A computational model (or a problem model) is a tuple

$$M = (\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB})$$

where

(1) $\text{Ent} = (A_1, A_2, \ldots, A_n, \ldots)$ is the sequence of the names of entities; the empty entity Omega is accepted; a number of entities are declared as *functional*;

(2) Ent_{func} is a list of the names of all functional entities with their types; the type of a functional entity F is defined as an expression of the form

$$(X \to Y)$$

where X and Y are lists of the names of entities from Ent; the list X is called the *argument* for F;

(3) DomEnt is a sequence of domains of entities from Ent; the domain of the entity A is denoted by $\text{Dom}(A)$; for the empty entity Omega the set Dom(Omega) is empty; for a list X. $\text{Dom}(X)$ is the Cartesian product of domains of all entities from the list X; a function (program) mapping $\text{Dom}(X)$ into $\text{Dom}(Y)$ is said to be the function (program) of the type $(X \to Y)$; for a functional entity F of the type $(X \to Y)$. $\text{Dom}(F)$ is to be the set of all programs of the type $(X \to Y)$;

(4) KB is a set of relations between entities from Ent.

We represent the forms of relations below.

2.2. *Forms of Dependencies*

One can correlate a *scheme dependency D* with a *real* relation D_{real}. We consider the following dependencies.

(1) A *functional dependency*

$$G : (X \to Y)$$

(here G is a functional symbol, and X and Y are lists of names from Ent) expresses on the *scheme*, or *structural*, level a relation of the form

$$Y = g(X)$$

where g is a function (or functional of higher order if names of functional entities are contained in the lists X and Y); g is called a realization for G. For example, the relation

$$s1 = \int_0^{t1} v(t)\, dt$$

could be expressed on the scheme level as a functional dependency

$$I : (v, t1 \to s1),$$

where I is the functional symbol, and v is the functional entity of the type $(t \to \text{val}\, v)$. If t is time, then $\text{val}\, v$ is the current value of v.

(2) An *operator dependency* [7] (or a conditional computability statement [3])

$$G:((X_1 \to Y_1),(X_2 \to Y_2),\ldots,(X_m \to Y_m) \to Y)$$

is treated as the functional dependency

$$G:(F_1,F_2,\ldots,F_m \to Y),$$

where $F_1,F_2,\ldots,F_m$ are functional entities of types $(X_1 \to Y_1),(X_2 \to Y_2),\ldots,(X_m \to Y_m)$ respectively. The integral above can be expressed as an operator dependency

$$I:((t \to \text{val}\, v),t1 \to s1)$$

(3) A *circular*, or *symmetrical dependency* [1, 8]

$$G:(X \to \text{Circ}(Y))$$

represents on the scheme level the equation

$$g(X,Y)=0,$$

which is solvable with respect to each variable B from the list Y; g is called a realization for G. For example, the law of cosines

$$c^2=a^2+b^2-2ab\cos\gamma$$

could be expressed as circular dependency

$$G:(a,b \to \text{Circ}(c,\text{gamma})).$$

(4) A *variant dependency* [9]

$$(Q,G_1,G_2):(X \to Y_1 \text{ OR } Y_2),$$

where Q is a predicate symbol and G_1 and G_2 are functional symbols, expresses on the scheme level the branching

$$\text{if } q(X) \text{ then } Y_1=g_1(X) \text{ else } Y_2=g_2(X).$$

The predicate q and functions g_1 and g_2 are called realizations for the predicate symbol Q and the functional symbols G_1 and G_2 respectively. The lists Y_1 and Y_2 are called alternatives. For example, the relation

$$\text{if } d>0 \text{ then sqd}=(d)^{0.5} \text{ else print}=\text{'there is no root'}$$

could be expressed on the scheme level as a variant dependency

$$(Q,G_1,G_2):(d \to \text{sqd OR print}).$$

It should be pointed out that by means of such variant dependencies a situation can be described where there are two sets of dependencies KB_1 and KB_2, and for some values of a list of parameters X the relations from KB_1 are valid, whereas for the other values of X the relations from KB_2 are valid. In this way a scheme dependency D is associated to the real dependency D_{real} of one of the above four kinds. Inversely, a real dependency D_{real} can be treated as a realization for the scheme dependency D.

It is convenient to introduce two levels (*real* and *scheme*) for computational models. A system KB_{real} of real relations can be expressed on the scheme level as a system KB of scheme dependencies. And inversely, the system KB_{real} of real relations can be treated as a realization for the system KB of scheme dependen-

cies. Therefore we can consider a scheme computational model

$M = (\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB})$

and a real computational model

$M_{\text{real}} = (\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB}_{\text{real}})$.

2.3. States. Samples

We introduce the undefined symbol undef.

A sequence

$s = (a_1, a_2, \ldots, a_i, \ldots, a_n, \ldots)$,

where for every i a_i is from $\text{Dom}(A_i)$ or equals undef, is said to be a *state* of the object. Every a_i is denoted also by $A_i(s)$ and is treated as

the value of the entity A_i in the state s.

For a list $X = B_1, B_2, \ldots, B_m$, we denote

$X(s) = (B_1(s), B_2(s), \ldots, B_m(s))$.

If some $B_j(s)$ is equal to undef, we shall take $X(s) = \text{undef}$.

A *sample* is an arbitrary set of states. For example, the following table represents a sample. Here p is a program mapping t into $2t$:

t	val v	v	$t1$	$s1$
0	0	p	undef	undef
1	2	p	undef	undef
2	4	p	3	9
undef	undef	p	−3	9

2.4. Admissible Samples

A sample T is called *admissible* for a computational model

$M = (\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB})$

if

(1) T is consistent with all functional entities from Ent_{func},

(2) all the dependencies from KB are satisfied in T.

More precisely, a function g of the type $(X \to Y)$ is said to be observable on T if, for any state s from T in which $X(s)$ is defined (is not equal to undef), we have

$Y(s) = g(X(s)) \neq \text{undef}$.

A sample T is *consistent* with a functional entity F of the type $(X \to Y)$ if

(1) for any state s from T in which $F(s)$ is defined and $X(s)$ is defined we have

$$Y(s) = F(s)(X(s)) \neq \text{undef}$$

[let us recall that $F(s)$ is a program of the type $(X \to Y)$],

(2) for any list of entities S, if a function of the type $(S, X \to Y)$ is observable on T, then a function of the type $(S \to F)$ is observable on T.

A functional dependency

$G:(X \to Y)$

is said to be satisfied on a sample T if a function of the type $(X \to Y)$ is observable on T.

A circular dependency

$G:(X \to \mathrm{Circ}(B_1, B_2, \ldots, B_m))$

is said to be satisfied on a sample T if for any i and any

$x, b_1, b_2, \ldots, b_{i-1}, b_{i+1}, \ldots, b_m$

there exists b from $\mathrm{Dom}(B_i)$ such that for any state s from T in which

$X(s) = x, \quad B_1(s) = b_1, \quad B_2(s) = b_2, \ldots, \quad B_{i-1}(s) = b_{i-1}, \ldots,$

$B_{i+1}(s) = b_{i+1}, \ldots, \quad B_m(s) = b_m$

we have

$B_i(s) = b.$

A variant dependency

$(Q, G_1, G_2):(X \to Y_1 \text{ OR } Y_2)$

is said to be satisfied on a sample T if there exists a predicate q on $\mathrm{Dom}(X)$ and a function g_1 of the type $(X \to Y_1)$ and a function g_2 of the type $(X \to Y_2)$ such that for any state s from T in which $X(s)$ is defined we have

(1) if $q(X(s))$ is true then $Y_1(s) = g_1(X(s))$ and is defined,

(2) if $q(X(s))$ is false then $Y_2(s) = g_2(X(s))$ and is defined.

3. SEMANTICS OF COMPUTATIONAL TASKS

In this section the acute semantics of a solvable computational task is given.

3.1. Formal Definition of Computational Task

A *computational task* is a tuple

$\mathrm{Task} = (M;\ X \Rightarrow Z)$

where M is a computational model, X is a list of inputs, and Z is a list of outputs.

3.2. A Very Nonconstructive Semantics of Computational Tasks

A computational task

$\mathrm{Task} = (M;\ X \Rightarrow Z)$

is said to be *solvable in the nonconstructive sense* if for any sample T which is admissible for M the functional dependency $H:(X \to Z)$ is satisfied on T.

We can give here three versions of unsolvability:

(1) A computational task Task is called *unsolvable in the first sense* if there exists a sample T such that T is admissible for M and for some state s from T, $X(s)$ is defined and $Z(s) = \text{undef}$. In this version the explicit "undef" is used.

(2) If there is no symbol "undef" in a sample T, then the sample T is called *total*. A computational task Task is called *unsolvable in the second sense* if there exists a total sample T such that T is admissible for M and for some states s_1 and s_2 from T we have

$$X(s_1) = X(s_2) \quad \text{and} \quad Z(s_1) \neq Z(s_2).$$

In this version "undefined" means a lack of single-valuedness.

(3) A computational task Task is called *unsolvable in the third sense* if there exists a total sample T such that T is admissible for M and for some infinite sequence of different states

$$s_1, s_2, \ldots, s_i, \ldots$$

from T we have, for any different i and j, $X(s_i) = X(s_j)$ and $Z(s_j) \neq Z(s_j)$. In this version "undefined" means "infinite" lack of single-valuedness.

If for any entity A the set $\text{Dom}(A)$ is infinite, then the equivalence of these three definitions can be proved.

3.3. Extremely Constructive Semantics: Solvability as the Existence of a Program with Uniform Structure

Let

$$M_{\text{real}} = (\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB}_{\text{real}})$$

be a real computational model. We consider relations which follow naturally from the relations of KB_{real}. Namely,

(1) if a function h of the type $(X, Y_1 \to Y_3)$ is defined by the equation

$$h(x, y_1) = f(x, g(y_1)),$$

then the relation

$$Y_3 = h(X, Y_1)$$

is called a natural consequence of the pair of relations

$$Y_2 = g(Y_1) \quad \text{and} \quad Y_3 = f(X, Y_2);$$

(2) if for a function h of the type $(Y \to B)$ we have

$$g(y, h(y)) = 0$$

for all y from $\text{Dom}(Y)$, then the relation

$$B = h(Y)$$

is called a natural consequence of the relation

$$g(Y, B) = 0;$$

(3) if a function h of the type $(X \to Y)$ is defined by the equation

$$\text{if } q(x) \text{ then } h(x) = f_1(x, g_1(x))$$
$$\text{else } h(x) = f_2(x, g_2(x)),$$

then the relation

$$Y = h(X)$$

is called a natural consequence of the relations

$$Y = f_1(X, Y_1) \text{ and } Y = f_2(X, Y_2) \text{ and}$$
$$\text{if } q(X) \text{ then } Y_1 = g_1(X) \text{ else } Y_2 = g_2(X);$$

(4) for a functional entity F of the type $(X \to Y)$, the relation

$$Y = h(F, X)$$

is called a natural consequence of $\mathrm{KB}_{\mathrm{real}}$, where the function h of the type $(F, X \to Y)$ is defined by the equation

$$h(p, x) = p(x)$$

for all p from $\mathrm{Dom}(F)$ and x from $\mathrm{Dom}(X)$;

(5) if for a functional entity F of the type $(X \to Y)$, a list of entities S, a function h of the type $(S \to F)$, and a function g of the type $(S, X \to Y)$, we have

$$h(s_0)(x) = g(s_0, x)$$

for all s_0 from $\mathrm{Dom}(S)$ and x from $\mathrm{Dom}(X)$ [recall that $h(s_0)$ is a program of the type $(X \to Y)$], then the relation

$$F = h(S)$$

is called a natural consequence of the relation

$$Y = g(S, X);$$

(6) such operations as the introduction of fictitious variables, the formation of vector functions, and the projection of vector results are permitted.

Then we take the transitive closure;

(a) Each relation of $\mathrm{KB}_{\mathrm{real}}$ is called a natural consequence of $\mathrm{KB}_{\mathrm{real}}$.

(b) If $D_1, D_2, \ldots, D_k$ are natural consequences of $\mathrm{KB}_{\mathrm{real}}$ and the relation D is a natural consequence of relations $D_1, D_2, \ldots, D_k$ in the sense above, then D is called a natural consequence of $\mathrm{KB}_{\mathrm{real}}$.

A program p of the type $(X \to Z)$ is said to be a solution for a computational task

$$\mathrm{Task}_{\mathrm{real}} = ((\mathrm{Ent}, \mathrm{Ent}_{\mathrm{func}}, \mathrm{DomEnt}, \mathrm{KB}_{\mathrm{real}}); X \Rightarrow Z)$$

if the relation

$$Z = p(X)$$

is a natural consequence of $\mathrm{KB}_{\mathrm{real}}$.

One may want the structure of a program p to be defined completely by the scheme descriptions for relations from $\mathrm{KB}_{\mathrm{real}}$. For this purpose we introduce the

concept of a scheme program. By a scheme program we mean an ordinary program in a programming language (with calls, conditional operators, etc.) in which

(1) each call or let operator of the form

$$Y = g(X)$$

is replaced by a scheme operator of the form

$$Y = G(X)$$

where G is a functional symbol or the name of a scheme procedure;

(2) each condition of the form

$$q(X)$$

is replaced by a scheme condition of the form

$$Q(X)$$

where Q is a predicate symbol;

(3) calls for some standard procedures, such as equation solvers or selectors, are accepted.

A scheme program p of the type $(X \to Z)$ is said to be a solution for a computational task

$$\text{Task} = ((\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB}); X \Rightarrow Z)$$

if for any realization KB_{real} for the system of (scheme) dependencies KB, the relation

$$Z = p(X)$$

follows naturally from relations of KB_{real}, where the program p is obtained by replacing all the functional and predicate symbols in the scheme program P by correspondent realizations from KB_{real}.

Theorem 3.3. Let KB *contain functional, operator, circular, and variant dependencies. For every entity A let the set* $\text{Dom}(A)$ *be infinite. Then a computational task*

$$\text{Task} = (M; X \Rightarrow Z)$$

is solvable in the nonconstructive sense of Section 3.2 *if and only if there exists a scheme program P of the type* $(X \to Z)$ *which is a solution for* Task.

4. CALCULUS OF DEPENDENCIES OLD

For treating computational tasks and models of this kind the oblivious lossless dependency calculus OLD is used.

4.1. The Language

By formulas we mean expressions of the form

$$([S], (X' \to Y') \to E)$$

where S, X', and Y' are lists of names from Ent, and E is a list Y or an expression Circ(Y) or an expression Y_1 OR Y_2. Possible ordering for lists is neglected; e.g., the list A, B, A is declared equal to the list B, A, A.

Informally, this formula "expresses" that there is a dependency

$$N:(S, F' \to E)$$

where F' is a functional entity of the type $(S, X' \to Y')$.

A computational model

$$M = (\text{Ent}, \text{Ent}_{\text{func}}, \text{DomEnt}, \text{KB})$$

is represented in the language by a list of formulas Γ_M:

(1) A functional, circular, or variant dependency

$$N:(X \to E)$$

is represented in Γ_M by the formula

$$([\varnothing], X \to E)$$

(here $\varnothing$ denotes the empty list).

(2) A functional entity F of the type $(X \to Y)$ is represented in Γ_M by the formula

$$([\varnothing], F, X \to Y)$$

and the formula

$$([\varnothing], (X \to Y) \to F).$$

For example, if we have a single functional entity v of the type $(t \to \text{val } v)$ and KB contains a single dependency $I:(v, t_1 \to s_1)$, then Γ_M is the list of three formulas

$([\varnothing], v, t \to \text{val } v)$,

$([\varnothing], (t \to \text{val } v) \to v)$,

$([\varnothing], v, t_1 \to s_1)$.

4.2. *Oblivious Proofs. Inference Rules for* OLD

We use special oblivious rules.

An ordinary inference rule is described with the help of a modus

$$(r) \quad \frac{f_1 \quad f_2 \quad f_3}{f}.$$

If a list of formulas Γ contains the formulas f_1, f_2, and f_3, then the list of formulas Γ' obtained by adding the formula f to Γ as a new element is said to be the result of application of the rule (r) to the list Γ.

An oblivious inference rule is described with the help of a modus

$$(r') \quad \frac{f_1^{\text{delete}} \quad f_2}{f}.$$

If a list of formulas Γ contains the formulas f_1 and f_2, then the list Γ' obtained by

adding the formula f to Γ as a new element and *deleting* an occurrence of the formula f_1 in Γ (f_1 marked by delete) is said to be the result of application of the rule (r') to the list Γ.

A derivation (proof) from Γ_0 is a sequence of lists of formulas

$$\Gamma_0; \Gamma_1; \ldots; \Gamma_i; \Gamma_{i+1}; \ldots; \Gamma_t$$

such that Γ_{i+1} is the result of application of an inference rule to Γ_i. We say that a formula f is derived from Γ_0 if f is contained in Γ_i.

The inference rules of OLD.

Strengthening:

$$\frac{([S],(X\to Y,A)\to E)^{\text{delete}} \quad ([S,X'']\to A,W)}{([S],(X\to Y)\to E)}$$

(where X'' either is empty or is the list X itself).

Deleting:

$$\frac{([S],(X\to A)\to E)^{\text{delete}} \quad ([S,X'']\to A,W)}{([S]\to E)}$$

(where X'' either is empty or is X itself).

Strengthening Circ:

$$\frac{([S]\to \text{Circ}(Y,A))^{\text{delete}} \quad ([S]\to A,W)}{([S]\to \text{Circ}(Y))}.$$

Deleting Circ:

$$\frac{([S]\to \text{Circ}(A))^{\text{delete}}}{([S]\to A)}.$$

Splitting:

$$\frac{([S]\to Y_1 \text{ OR } Y_2) \quad ([S,Y_1]\to A,W_1) \quad ([S,Y_2]\to A,W_2)}{([S]\to A)}.$$

Weakening:

$$\frac{([\varnothing],(X\to Y)\to E)}{([S],(X\to Y)\to E)}.$$

Introduction:

$$\overline{([S]\to S)}.$$

Empty:

$$\frac{([S]\to \text{Omega},W)}{([S]\to A)}.$$

Conjunction:

$$\frac{([S] \to Y_1)^{\text{delete}} \quad ([S] \to Y_2)^{\text{delete}}}{([S] \to Y_1, Y_2)}.$$

Here A is a name of entity from Ent; $S, X, Y, Y_1, Y_2, W, W_1, W_2$ are lists of names from Ent; E is a list Y' or an expression Circ(Y') or an expression Y_1 OR Y_2; and in the strengthening rules Y is required to be nonempty.

4.3. *Correctness of* OLD

Theorem 4.3. If a formula

$$([X] \to Z)$$

is derived from Γ_M *in the calculus* OLD, *then the computational task*

$$(M;\ X \Rightarrow Z)$$

is solvable in any meaning.

4.4. *Constructivity of* OLD

Theorem 4.4. According to an arbitrary derivation of a formula $([X] \to Z)$ *from* Γ_M *in* OLD *one can assemble (in the natural way) a scheme program P of type* $(X \to Z)$ *such that P is a solution for the computational task*

$$(M;\ X \Rightarrow Z).$$

4.5. *Strong Completeness of* OLD

None of the rules except for the weakening and introduction rules introduce new lists S. Let us restrict the weakening and introduction rules. Let

$$X_1, X_2, \ldots, X_m$$

be sequence of all *different* arguments for functional entities from Ent_{func} and alternatives from variant dependencies of KB. By $\text{In}_{M,X}$ we denote the set of all lists S of the form

$$S = X, X_{i1}, X_{i2}, \ldots, X_{ir}$$

where $0 \le r \le m$ and $1 \le i1 < i2 < \cdots < ir \le m$. A parameter S in the weakening and introduction rules should be selected from the set $\text{In}_{M,X}$.

A derivation

$$\Gamma_0; \Gamma_1; \ldots; \Gamma_i; \ldots; \Gamma_t$$

from the list Γ_M in OLD is said to be X-complete if all deductive power of the inference rules has been exhausted, namely,

(1) the rules for strengthening, deleting, strengthening Circ, deleting Circ, and conjunction are not applicable to Γ_t;

(2) if the splitting or empty rule with given parameters S and A is applicable to Γ_t, then Γ_t contains a formula of the form $([S] \to A, W)$;

(3) for any S from $\mathrm{In}_{M,X}$ the list Γ_t contains a formula of the form $([S] \to S, W)$;

(4) for any S from $\mathrm{In}_{M,X}$ and each formula $([\varnothing], (X' \to Y') \to E)$ from Γ_0, some Γ_i contains a formula of the form $([S], (X' \to Y') \to E)$.

Theorem 4.5. Let $\mathrm{Dom}(A)$ *be infinite for each nonempty entity* A. *Let*

$$\Gamma_0; \Gamma_1; \ldots; \Gamma_t$$

be an arbitrary X-complete derivation from Γ_M *in* OLD. *Then if* Γ_t *does not contain a formula of the form*

$$([X] \to Z, W),$$

then the computational task

$$\mathrm{Task} = (M;\ X \Rightarrow Z)$$

is unsolvable in the sense of Section 3.2, *and there is no scheme program P which is a solution for* Task.

5. COMPLEXITY OF THE ALGORITHM OF ANALYSIS AND SYNTHESIS

5.1. *The Algorithm Based on the Calculus* OLD

Let us consider the following algorithm:

Input. A computational task

$$\mathrm{Task} = (M;\ X \Rightarrow Z)$$

Output. A scheme program P of the type $(X \to Z)$ which is a solution for Task if Task is solvable, or an explanation of the nature of unsolvability otherwise.

Method. An X-complete derivation from Γ_M in OLD is constructed. If a formula of the form $([X] \to Z, W)$ is observed in the derivation, then with the help of Theorem 4.4 the derivation is transformed into the scheme program P of the type $(X \to Z)$. Otherwise, with the help of Theorem 4.5, a counterexample is constructed.

Theorem 5.1. Let $\mathrm{Dom}(A)$ *be infinite for every* A. *Then the algorithm above runs correctly on all computational tasks.*

5.2. *Subquadratic Space*

On the basis of the strong completeness of OLD one can implement the algorithm of analysis and synthesis in a small space.

Theorem 5.2 [9]. A computational task $(M;\ X \Rightarrow Z)$ *can be analyzed and its solution synthesized in space*

$$O(l) + O(nd)$$

where l is the number of all occurrences of entity names in $\mathrm{Ent}_{\mathrm{func}}$ *and* KB, *n is the number of different entity names from* $\mathrm{Ent}_{\mathrm{func}}$ *and* KB, *and d is the number of different alternatives in* KB.

Corollary 5.2. If there is no variant dependency in KB, *then one can analyze a computational task* $(M;\ X \Rightarrow Z)$ *and synthesize its solution in linear space*

$$O(l)$$

5.3. Degree of Interaction of Subtasks

The problem of analysis of computational tasks $(M;\ X \Rightarrow Z)$ turns out to be PSPACE-complete even if KB contains *only* functional dependencies. Therefore all known algorithms of analysis and synthesis are forced to perform an exponential search for "almost all" computational tasks. In spite of this, all examples of "bad" computational tasks are unnatural from the programmer's point of view.

We introduce some level (rank) of interaction of computational tasks, so that natural (realistic) tasks have a small level of interaction.

In performing the computational task $(M;\ X \Rightarrow Z)$ there may be *subtasks*, e.g.

(1) Using a functional dependency of the form

$$G : (F \to Y)$$

where F is a functional entity of type $(X' \to Y')$, a subprogram of the type $(X' \to Y')$ named by F has to be synthesized; in other words, some subtask of the form $(M';\ X' \Rightarrow Y')$ has to be performed. The input of this subtask is the argument X' for F.

(2) Using a variant dependency

$$(Q, G_1, G_2) : (X' \to Y_1 \text{ OR } Y_2)$$

for computing some Z' from X', it is supposed that two branches are synthesized: First, some subtask of the form $(M';\ Y_1 \Rightarrow Z')$ is performed; the input of this subtask is the alternative Y_1. Secondly, some subtask of the form $(M';\ Y_2 \Rightarrow Z')$ is performed; the input of this subtask is the alternative Y_2.

In performing the main task, subtasks can interact; namely, we can perform a subtask provided that values of inputs of some other subtasks are given in addition. Embedding of procedures in programs is related to this phenomenon. The maximal number of subtasks with *different* inputs which can interact in the process of performing the main task at a moment is called the degree of subtask interaction or degree of strong embedding of procedures.

We give precise definitions.

Let

$$X_1, X_2, \ldots, X_m$$

be the sequence of all different arguments of functional entities from $\mathrm{Ent}_{\mathrm{func}}$ and alternatives from KB. By $\mathrm{In}'_{M,X}$ we denote the set of all lists S of the form

$$S = X, X_{i1}, X_{i2}, \ldots, X_{il}$$

where $0 \le l \le r$ and $1 \le i1 \le i2 \le \cdots \le i1 \le m$. Each such list S is a conjunction of the input X of the main task and inputs $X_{i1}, X_{i2}, \ldots, X_{il}$ of l possible subtasks.

A sample T is said to be consistent with a functional entity F of the type $(X' \to Y')$ down to depth $r+1$ if

(1) for any state s from T in which $X'(s)$ is defined and $F(s)$ is defined we have

$$Y'(s) = F(s)(X'(s)) \neq \text{undef};$$

(2) for any list of entities S from the set $\text{In}^r_{M,X}$, if a function of the type $(S, X' \to Y')$ is observable at T, then a function of the type $(S \to F)$ is observable at T.

A sample T is said to be consistent with a variant dependency

$$(Q, G_1, G_2) : (X' \to Y_1 \text{ OR } Y_2)$$

down to depth $r+1$ if, for any list S from the set $\text{In}^r_{M,X}$ for which a function of the type $(S \to X')$ is observable at T, and for any list of entities Z', if functions of types $(S, Y_1 \to Z')$ and $(S, Y_2 \to Z')$ are observable at T, then a function of the type $(S \to Z')$ is observable at T.

A sample T is called admissible for the computational model M down to the depth r with the common input X if

(1) T is consistent down to the depth r with all functional entities from Ent_{func},

(2) T is consistent down to the depth r with all variant dependencies from KB,

(3) all functional and circular dependencies from KB are satisfied on T.

We shall say that the computational task

$$\text{Task} = (M;\ X \Rightarrow Z)$$

is solvable (in the nonconstructive meaning) with degree r of subtask interaction if for any sample T which is admissible for the model M down to the depth r the functional dependency

$$H : (X \to Z)$$

is satisfied on T.

Theorem 5.3. A computational task

$$\text{Task} = (M;\ X \Rightarrow Z)$$

is solvable with degree r of subtask interaction if there exists a scheme program P such that P is a solution for Task *and the maximal depth of embedding of subprograms (procedures) and conditional operators does not exceed r.*

5.4. *Quasipolynomial Time*

Theorem 5.4 [9]. One can analyze a computational task

$$\text{Task} = (M; X \Rightarrow Z)$$

and synthesize its solution in time

$$O\left(\frac{m^{3r}}{(r!)^3}n^2 l\right)$$

and in space

$$O(l) + O\left(n\frac{m^r}{r!}\right),$$

where

l is the number of all occurrences of entity names in Ent_{func} *and KB,*

n is the number of different names from Ent_{func} *and KB.*

m is the number of different arguments for functional entities from Ent_{func} *and alternatives from variant dependencies of KB,*

r is the minimal degree of subtask interaction with which the task Task is solvable.

Corollary 5.4. In particular, for $r = 1$ *we obtain a quadratic time synthesis algorithm for solving computational tasks with separable subtasks and common input* [7]. *This class of tasks is larger than the class of tasks with independent subtasks from* [4].

So, treating computational tasks on the basis of our lossless calculus OLD, an exponential execution time should be expected *in the worst case*. But this phenomenon arises from very unnatural tasks for which extreme cross-linking of all possible subtasks is necessary. Our synthesizer runs in polynomial time (and almost linear space) on natural computational tasks; the degree of the polynomial is determined by the minimal depth of embedding of subtasks which is achieved in any solution for the main task.

6. COMPLEXITY OF THE SYNTHESIS OF OPTIMAL PROGRAMS

6.1. Cleaning the Synthesized Programs

There may be "dead" entities in a synthesized program which are not needed for computing the values of outputs of the task [2, 5].

Theorem 6.1. On the basis of the calculus OLD *one can synthesize programs which are free from "dead" evaluations. This algorithm for the synthesis of clean programs can run in quasipolynomial time and subquadratic space according to Theorems* 5.2 *and* 5.4.

6.2. The Synthesis of Programs with the Minimal Sequential Execution Time

Theorem 6.2 [11]. Consider the simplest case: there is no functional entity in the computational model M (Ent_{func} *is empty*), *and* KB *contains only functional*

dependencies. Then the problem of constructing a program for a computational task $(M;\ X \Rightarrow Z)$ *with the mimimal sequential execution time is NP-complete.*

As a corollary we have that an arbitrary algorithm which synthesizes programs with the minimal sequential execution time is forced (most probably) to run in exponential time.

6.3. *The Synthesis of Programs with the Minimal Parallel Execution Time*

This problem is easier. Generally, difficulties arise mainly because calculi are not deterministic. But, in view of the fact that our calculus OLD is *strongly* complete, nondeterministic and free search can be used. Without loss of correctness, completeness, and efficiency, various strategies for derivation search can be utilized to synthesize the best possible programs. We give such an example.

Theorem 6.3 [11]. *Let* $\mathrm{Ent}_{\mathrm{func}}$ *be empty and* KB *contain functional and circular dependencies only. Then in linear time*

$$O(l)$$

one can construct a program for a computational task $(M,\ X \Rightarrow Z)$ *with the minimal parallel execution time.*

In the theorem the execution time of a module in the synthesized program is assumed equal to 1. If a system of positive "weights" for dependencies from KB is given, then in almost linear time

$$O(l \log_2 n)$$

one can synthesize a program with the minimal weighted parallel execution time. Here l is the number of all occurrences of entity names in KB, and n is the number of different entity names for KB.

In conclusion it should be pointed out that on the basis of similar lossless calculi one can obtain polynomial (and even linear or subquadratic) algorithms also for

(1) the membership problem in the theory of relational data bases with functional and multivalued dependencies,

(2) recognizing derivability of formulas of some kind in the classical and intuitionistic propositional calculi,

(3) recognizing the validity of Horn formulas in the one-place predicate logic,

(4) flow analysis of AND-OR graphs, etc.

REFERENCES

1. Kahro, M. I., Kalin, A. P., and Tyugu, E. H., *The Instrumental Programming System for* RYAD *Computers* (PRIZ) (in Russian), Moscow, 1981, p. 157.
2. Tyugu, E. H., *Knowledge Based Programming*, Addison-Wesley, 1987.
3. Mints, G. and Tyugu, E., Justification of the structural synthesis of programs, *Sci. Comput. Programming*, 1982, No. 2, pp. 215–240.

4. Mints, G. and Tyugu, E., Structural synthesis and Nonclassical Logics, in: *The Third Conference on Application of Methods of Mathematical Logic*, Tallinn, 1983, pp. 52–60.
5. Babaev, O. I., Lavrov, S. S., et al., Spora as a System of Programming with the Automatic Synthesis of Programs, in: *The Third Conference on Application of Methods of Mathematical Logic*, Tallinn, 1983, pp. 29–41.
6. Lavrov, S. S., The Architecture of Knowledge Bases, in: *Software for Computers with new Architectures* (in Russian), Novosibirsk, 1986.
7. Dikovski, A. and Kanovich, M., Computational Models with Separable Problems (in Russian), *Tekh. Kibernet.* No. 5, 1985, pp. 36–59.
8. Kanovich, M. I., Efficient Logical Algorithms of Analysis and the Synthesis of Dependencies, *Soviet Math. Dokl.* 32(3):867–871 (1985).
9. Kanovich, M. I., Quasipolynomial Algorithms for Recognizing the Satisfiability and Derivability of Propositional Formulas, *Soviet Math. Dokl.*, 34(2):273–277 (1986).
10. Kanovich, M. I., *Logical Methods for the Synthesis of Programs* (in Russian), Kalinin, 1986, p. 44.
11. Kanovich, M. I., Efficient Calculi as a Tool for Reducing Search. *Voprosy Kybernet. (Moscow)* BK–131:149–167 (1987).